CHARTER MARK

MAY 0 6 2002 JC87 PATENT & TRADEMARK OFFICE

PATENTS • DESIGNS The Patent Office COPYRIGHT • TRADE MARKS

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

CERTIFIED COPY OF PRIORITY DOCUMENT

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed [illegible]

Dated 18 February 2002

BEST AVAILABLE COPY

Patents Form 1/77

Patents Act 1977
(Rule 16)

The Patent Office

1/77

THE PATENT OFFICE
A
14 FEB 2001
NEWPORT

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

The Patent Office

Cardiff Road
Newport
Gwent NP9 1RH

1. Your reference	HL78193/000/CIV	
		15FEB01 E606251-3 D02859 P01/7700 0.00-0103687.0
2. Patent application number *(The Patent Office will fill in this part)*	14 FEB 2001 **0103687.0**	
3. Full name, address and postcode of the or of each applicant *(underline all surnames)*	PIXELFUSION LIMITED Wallscourt Farm Filton Road Bristol BS34 8RB	
Patents ADP number *(if you know it)* If the applicant is a corporate body, give the country/state of its incorporation	United Kingdom	07515638003
4. Title of the invention NETWORK PROCESSING - ARCHITECTURE II		
5. Full name of your agent *(if you have one)*	Haseltine Lake & Co.	
"Address for service" in the United Kingdom to which all correspondence should be sent *(including the postcode)*	Imperial House 15-19 Kingsway London WC2B 6UD	
Patents ADP number *(if you know it)*	34001	✓

	Country	Priority application number *(if you know it)*	Date of filing *(day/month/year)*
6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and *(if you know it)* the or each application number			

	Number of earlier application	Date of filing *(day/month/year)*
7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application		

8. Is a statement of inventorship and of right to a grant of patent required in support of this request? *(Answer "Yes" if:* *a) any applicant named in part 3 is not an inventor, or* *b) there is an inventor who is not named as an applicant, or* *c) any named applicant is a corporate body.* *See note (d))*	Yes

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

Continuation sheets of this form	-
Description	30
Claim*(s)*	-
Abstract	-
Drawing*(s)*	-

10. If you are also filing any of the following,

Priority documents	n/a
Translations of priority documents	n/a
Statement of inventorship and right to a grant of patent *(Patents Form 7/77*	-
Request for preliminary examination and search *Patents Form 9/77)*	-
Request for substantive examination *(Patents Form 10/77)*	-
Any other documents *(please specify*	-

11. I/We request the grant of a patent on the basis of this application

Signature [illegible] Date 14 February 2001

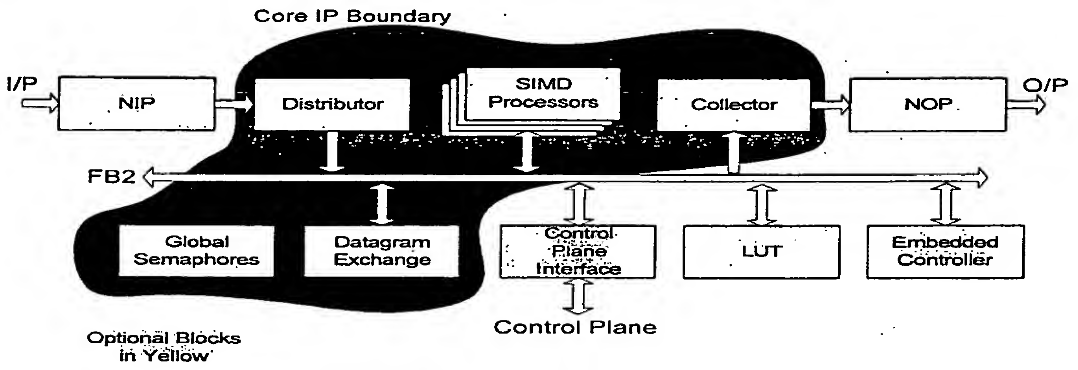
12. Name and daytime telephone number of person to contact in the United Kingdom Mr. Chris Vigars [0117] 9103200

Warning
After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes
a) If you need help to fill in this form or you have any questions, please contact the Patent Office on 0645 500505.
b) Write your answers in capital letters using black ink or you may type them.
c) If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.
d) If you have answered "Yes" Patents Form 7/77 will need to be filed.
e) Once you have filled in the form you must remember to sign and date it.
f) For details of the fee and ways to pay please contact the Patent Office.

1 Summary

The aim of this document is to stand back a bit from the current architecture proposal and re-shape it from the PoV of digital system design using IP blocks. This grew out of the desire to support multiple processing rounds per packet in a clean way by avoiding packet wrapping between SIMD processors, and well, one thing leads to another... The architecture concept remains broadly the same, but the system is partitioned and recombined in ways which the authors believe better meet the design targets.
We've tried to form as concise and complete description as possible to avoid just adding noise to the discussion space, so the pages that follow contain a lot of detail about structure and operation. This section provides an overview of the proposal and how it differs from the existing one.



The architecture consists of a core set of IP blocks, to which are added application specific blocks for particular functions or interfaces. The core architecture is general purpose – it contains no hardware specifically for network or packet processing, and can be used for the processing of any stream based data, to which DSP processors or dedicated hardware would be traditionally applied e.g. audio/video, radar/sonar, cellular channel coding etc.. It is, in effect, a general SIMD dataflow processing architecture.
Functions that are unique to an application like packet processing are contained in plug-in blocks. These may deal with the insertion or extraction of data in particular formats (NIP, NOP & CPI) or hardware acceleration for certain tasks (LUT).

We believe this architecture has the following advantages:

1. More like platform IP and less like a chip design.
2. Follows standard conventions of digital system and processor design.
3. Software is king - functionality & data paths are not cast in hardware but programmed in a uniform and consistent manner, including packet I/O, recirculation of packets from one SIMD processor to another, and exchanging packets with the control plane.
4. The dedicated packet interface blocks (NIP, NOP) are radically simplified, with much of their previous function being rolled into general purpose and reusable Distributor/Collector blocks.
5. Because of the modularity and consistency of operation, several of the blocks (Distributor, Collector, Datagram Exchange, Control Plane Interface) are constructed from different

combinations of common sub-components, allowing acceleration of block development through reuse of those sub-components.
6. Extension of the architecture to flow-based packet processing such as encryption/decryption of fragmented packets and flow classification/routing will require a shared memory model to store flow state. This architecture has that model in from the start in the form of the Datagram Exchange block.

The modular nature of the architecture means it can be developed in a number of stages. At the end of the document we describe a five stage programme. Stage one has the basic fast path processing and might be a suitable candidate for the April architecture deliverable.

2 Design Principles

Architecture is driven by a set of conflicting guidelines, priorities and principles which influence how design choices are made. These are the principles we have followed in defining this design, together with their implications for the architecture.

Principle	Implication
PixelFusion is NOT designing a network processor chip. We are designing IP to be deployed in customer chip designs.	Modular, Scalable - a set of IP blocks that do their own thing and can interlock with each other without knowing each others functional details or internal state. That ensures that those blocks can be put together in flexible and scalable ways without some complex interaction 'breaking' the operation of the system in ways which are only apparent under detailed simulation.
Our IP components will co-exist with other vendors components on the same SoC.	Interoperable - IP blocks that can slot in with another vendor's components with a clean interface, that conforms to industry practise & standards in assembling digital systems. For instance, we have chosen to comply with the VCI spec for block to block communication.
Our IP is a software based platform technology.	Express functionality in software running on general purpose hardware, not directly expressed in dedicated hardware. Intelligent processor blocks that run software are the 'masters' in the system and dumb hardware blocks are the 'slaves'.
The architectural components are independent of the application.	The basic set of IP blocks are general purpose and useable in a variety of data processing systems. Any hardware specifically for an application is cast in the form of an optional plug-in.
Power consumption is the most severe system design constraint for the functionality and performance required.	Architecture is optimised for minimum power consumption. - Minimum wastage of memory. Intelligent use of hierarchy - No large structures infrequently used (logic or interconnect) Data to travel the minimum distance through the system.

2.1 Building a System

In addition to the above principles we have another given: PixelFusion technology is based around massively parallel SIMD processors (abbreviated to SP in this document). How should we go about defining our IP products in terms of a family of components from which customers can assemble network processor devices (or any other classes of systems)?
One. Let's start with the SIMD processor. As processors go, it's fairly unusual. On F150 we had the luxury of making it as unusual as we wanted it to be, and we did. As an IP offering we should take a different approach: make our SP product look as un-scary as we can - as much like a 'normal' processor as possible. That way system designers can pick it up and sketch out how a system is going to work without having to absorb a lot of unfamiliar stuff. We should hide the SIMD nature from the system architect, or where that's not possible, we present the weird stuff in terms he already understands. What does a processor do?

1. Fetches and executes a program.
2. Reads data in from outside.
3. Stores it for a while, and processes it.
4. Writes out processed data.
5. Responds to signals (interrupts).

Two. Now consider the target application space – processing of a stream of data, network data packets for example.. We have a stream of data arriving at a port. We can't hold up that stream – it just pours in and we have to deal with it. That's a 'real time' requirement. The task is to get that data into the processor(s), do work on it, and then send it out again in another stream. The output is non-real time in the sense that we don't have to conform to a fixed schedule for sending the data, but can send it when it is ready, preserving the same order it arrived in (but perhaps with some kind of output flow control).
We chose to deploy multiple SPs to better match the uniform data flow and to decrease latency through the system. We know that the processors will need access to special purpose hardware accelerator blocks (e.g. LUT, Counter block) as part of their processing. The system also needs to send and receive data to and from an auxiliary system (the host or control plane) over some kind of interface. What we have not defined is exactly what kind of tasks or algorithms the processors are going to perform, and at this stage we don't want to. That should be entirely down to the application software guy to figure out – to exploit the power of the hardware however he sees fit.

Three. Define the system in such a way that supplies the maximum value to the software guys while maintaining a familiar feel to the system hardware guys. We will need some hardware to manage the data flow in and out (like a DMA engine), and to distribute and collect data to and from the processors. For communication between processors (if the software warrants it) and for control plane interface, a shared memory model is conventional, consistent and gives good flexibility. Control and synchronisation of the system components can be programmed through a set of shared hardware semaphores. The various components are connected by a common bus system that supports a shared address space.
These are familiar concepts for hardware, software and systems engineers – the hardware provides low level functionality and performance that is general purpose and consistent, while the software uses these low level functions to define what the system actually does. The special function hardware for any application is pulled in from a library of discrete 'plug-in' blocks with consistent interfaces (normally called 'peripherals'), which do not alter the basic architecture of the system.

Four. Draw a clean 'boundary' around the system to form a self-contained processing 'subsystem'. This subsystem fits into the overall customer architecture with standard and familiar interfaces. The subsystem itself can be configured or scaled internally without disruption of the larger system around it.

3 System Description

Using the approach described above we now construct a subsystem suitable for building network processors around SIMD processors. The simplified external view of the subsystem looks like the figure below. A number of these can be combined in a system or even on the same chip.

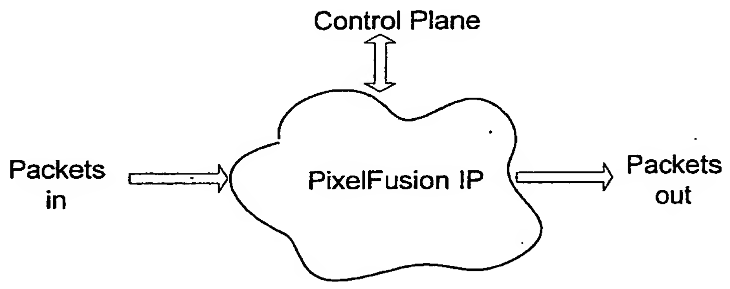


Figure 3.1: Simplified black box view of NP1

It is not essential that everything we define in our platform architecture goes into our first implementation or deliverable. Different levels of functionality can be provided in a staged development. What we do implement, however, must conform as a subset of the chosen eventual architecture. Once we are committed to delivering an architecture to customers we will not have the luxury of taking a few months out to redesign it.

In the sections below the structure and operation of the system is described. Not all of the blocks or functionality within blocks is required to form a working system. Section 5 shows a multi-stage implementation plan which starts with a basic system of limited but useful functionality and then extends it to cover more capabilities. The basic system may be adequate for the April 01 architecture deliverable.

In all of the descriptions below we discuss the FB2 bus structure as a logical interconnect. No physical implementation details such as topology or lane allocation should be inferred from the diagrams. It is assumed that where blocks contain significant amounts of memory (e.g. LUT) that the memory itself can be accessed as a naked resource, bypassing the block function. It is also assumed that such blocks can either use their own embedded memories or other memory entities in the system by means of the common bus system with global shared address space. The combination of these two principles provides complete associativity between block functions and memory resources, to realise a truly versatile IP platform from which a large number of system solutions can be devised.

3.1 General system model

Section 2 described some general principles for system design to ensure that components interact in a consistent, intuitive and robust way and that data flow is properly controlled. The most important principle is that processors negotiate for resources and manage the data flow to and from them. These resources are typically data buffers. The presence of buffers in data streams is important to de-couple the transfer rates and profiles of traffic produced (and consumed) by the processors which inevitably sit either side of each buffer. Data enters and leaves the system in a controlled manner via interfaces, with data entering the system always flowing from the interface to a data buffer block. In this model then we have the notion of three types of system entity: the processor, a data buffer, and an interface. The models of these entities should be representative of any block in the system which sits in the path of a data stream. Blocks may implement some or all of the interfaces and behaviours of their model types as described below.

Note: The models do not represent resources used by the processors which do not sit directly in the path of data. Examples of such 'Peripheral' blocks in the PixelFusion NP are the LUT, global semaphore block and counter block.

3.1.1 Data formats

This document identifies two domains within which data may flow - a system domain which contains all PixelFusion system IP components and plug-in blocks and the wider application domain external to the system.

Consider first the application domain. In most applications which handle streams of data the stream is rarely amorphous. Some kind of structure normally exists whereby data is grouped into frames, packets, cells etc. The natural data unit of the application domain is referred to generically as the datagram in this document. These datagrams may be of arbitrary and variable length.

Within the system domain, although we are still interested in the propagation of datagrams the system implementation can cause datagram fragmentation. Datagrams and datagram fragments transferred within the system domain are referred to as chunks. On entry into the system domain datagrams become single chunks and the chunk size is simply defined by the datagram size. Within the system domain processors are a common cause of chunk fragmentation into smaller chunks. All processors may have a natural size for handling blocks of data. For instance, the SIMD core chunk size is defined by the DIO buffer size configured into the IP. An embedded processor chunk may be determined by the cacheline size. Generally speaking, when processors read data in chunk form from data buffers the chunk sizes are defined by whatever format the processor requests the data in.

Chunking requires that a common control header be attached to all chunks transferred between entities in the system domain in order that the start and end of datagrams may be identified. Although at the system level datagram framing is the main purpose of the chunk header, it is recognised that system implementations can benefit from the presence of other control information. Taking the existing chunk header format and casting it in a more generic way the following structure is initially proposed:

Format of Chunk Header Word (64 bits)

Field	Subfield	Bits	Function
System flags	Common flags are used at the system level		
	Datagram start	1	Identifies a chunk containing the first bits of a datagram
	Datagram end	1	Identifies a chunk containing the last bits of a datagram
	Datagram wrap	1	Identifies chunks of datagram wrapped between requests
	Batch end	1	Identifies last chunk in a batch read
	Exception bit	1	Identifies a chunk for offline processing
	Error bit	1	Identifies a chunk which is incompatible with the system
	Reserved	2	For future system level control functions
Block specific bits	General purpose bits which may be defined by application specific hardware or software		
	Reserved	8	Flags reserved for local use by processors
	Chunk length	16	Useful to SIMD for identifying PE memory occupancy
	Undefined	32	May be defined arbitrarily by the application

The block specific bits reserved for local use by processors can be defined and used internally by any processor block. They should not be used to convey information between processors. For example, processors which have internal paths for data reprocessing may define a 'recirculation' bit to identify between chunks which must be output or chunks which must be recirculated after processing.

The undefined bits are intended to be used for inter-processor exchange of control information. For example, in networking, layer 2 packet protocol information extracted by the NIP block may be inserted into bits of this field. The identity of the physical line on which datagrams arrived could also be recorded.

3.1.2 Processor blocks

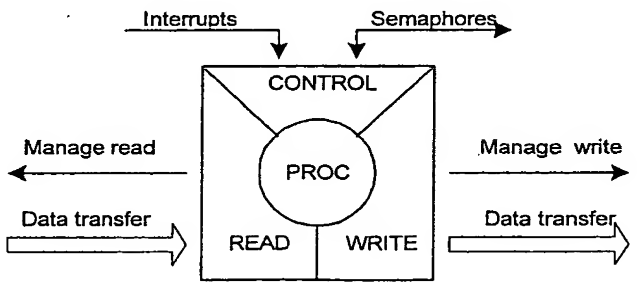


Figure 3.2: Basic system interfaces of generic processor model

CONTROL: Two mechanisms are provided for high level system control messaging and synchronisation.

> **Interrupts:** It is common in embedded control systems for interrupts to be used as a means by which passive system components can inform the controller that they have data which must be processed. The PF architecture is no different. Processors can be notified of the presence of datagrams in data buffers using interrupts.
>
> **Global semaphores:** An alternative to interrupts is polling. Multiple processors which poll shared resources can use general purpose global semaphores to synchronise their accesses. Global semaphores could also be deployed to resolve resource contention by providing mutual exclusivity to shared resources.

READ: Processors retrieve data on request only. Unsolicited data is never written directly to processors. There are two mechanisms for achieving this in the PixelFusion architecture - Conventional read and Batch read.

CONVENTIONAL READ: The conventional read follows the following standard format.

> **Manage read:** On receiving an interrupt a processor reads control registers in the interrupting block (normally a data buffer) to determine the location and size of a datagram which must be fetched from a memory. The chunk header for the datagram is also retrieved at this point. After the datagram has been subsequently read the processor writes back to the interrupting block to inform it that the data has been retrieved.
>
> **Transfer data:** The act of data transfer is simply a read transaction across the bus. Because the processor reads directly from a memory the conventional read transaction can only be used by processors accessing data buffer blocks whose data buffer is implemented as an addressable memory on the bus. Data buffer blocks with embedded hardware FIFOs cannot be accessed in this way. This is clarified in the data buffer block description.

Note that in conventional reads the datagram is effectively read as a single chunk directly from memory in one or more bus transcations. The logically attached chunk header is retrieved in the same transaction sequence but from a different physical location. This is perfectly legitimate. Chunk headers are held separately in all data buffer block implementations regardless of the location of the chunk header memory.

BATCH READ: The batch read enables processors to fetch multiple datagrams using a single request.

> **Manage read:** This is an instruction to a data buffer to transfer a specified quantity of data to the processor in a specific format. Conditions for the transfer are also provided. Parameters passed are therefore:
>
> - *Processor chunk size* – Maximum size for each chunk required by the processor.
> - *Chunks required* - Informs the data buffer how many chunks should be sent.
> - *Conditions* - This tells the data buffer how it should fill the chunks. This is described further in the next section.
>
> **Data transfer:** The fragmentation of datagrams into chunks is simply a consequence of the attributes of processors. The main role of chunk headers (as illustrated below) is simply to identify all chunks of a given datagram and mark the start and end. Little has changed from the original concept.

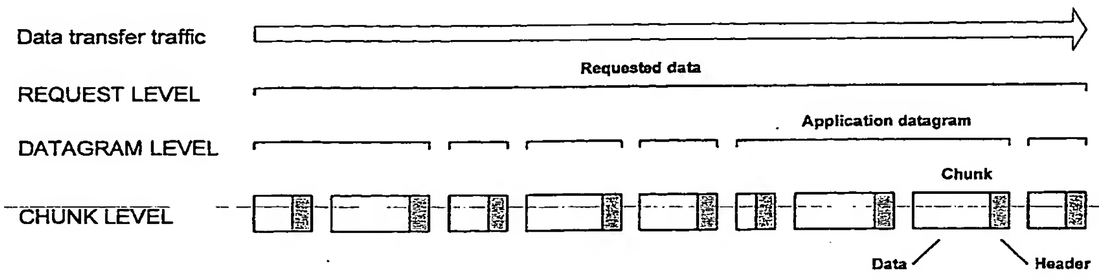


Figure 3.3: Organisation of data flowing from a data buffer to a processor during a batch read.

To recap then, datagrams are loaded in chunks which do not exceed the chunk size of the requesting processor..At the higher request level, the producer in the data buffer must make a decision based on the conditions supplied by the processor how to terminate the request. For instance, if wrapping is disallowed then the producer must terminate the request when the number of chunks of the next datagram exceeds the remaining 'chunk space' left in the processor.

WRITE: Processors send data to data buffers. Normally it is expected that the processor must negotiate for ownership of the intended destination before sending the data. However, write setup may not be required only if the destination is not shared.

Manage write: A handshake with the intended destination to ensure that it is ready to receive. This would typically be implemented using a hardware semaphore in the destination block .

Data transfer: The structure of output data can be totally arbitrary as far as the data buffer is concerned. Data is likely (though not required) to be output in chunk sized write bursts on the bus.

3.1.3 Data buffer blocks

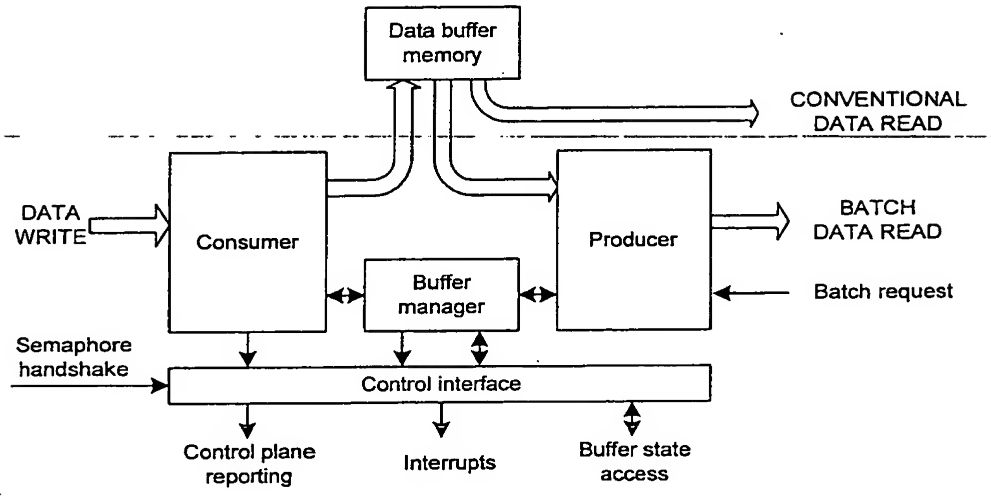


Figure 3.4: Basic structure of a fully endowed data buffer block

Data buffers are sources and sinks of data for processors. Any data buffer in the system should interact with other entities in the system via some or all of the basic interfaces shown in figure 2.4. The data read, write, request and control interfaces are all described in the processor section above. Components are described as follows:

CONSUMER: Receives chunks and identifies from the address supplied by the bus which of the one or more buffers in the data buffer memory to forward the data to. The data buffer block therefore owns an address aperture in system address space. Each configured buffer is assigned a single unique address from the pool. As chunks are written to the buffer the chunk header is removed. The first chunk header of each datagram is passed to the buffer manager. The consumer tracks chunks arriving for each buffer. When complete datagrams are seen to have been written to the buffer the buffer manager is informed.

Note that by allocating a buffer per data source, contention is avoided and stream order is maintained. Data chunks belonging to different streams may thus be interleaved on arrival at the consumer.

The consumer is configured with both blocking and non-blocking stream handling behaviours. Blocking behaviour simply allows data to back up onto the bus if buffers overflow. In non-blocking mode the consumer must tail drop from buffers which overflow and restore state information in the buffer manager for buffers which contain truncated datagrams.

DATA BUFFER MEMORY: Data buffer block instances which serve single streams of data may have buffer memories implemented as embedded hardware FIFOs. The buffer could alternatively be implemented as a memory - either embedded in the block or sitting on the on-chip bus. Any on or off-chip memory could therefore be used to provide the memory for a data buffer. As described in the previous section, if the buffer memory is addressable then conventional read accesses are supported. Buffers in memory are typically implemented as FIFO queues.

BUFFER MANAGER: Regardless of whether the data buffer block contains a single hardware FIFO queue or uses an external memory block as a data store, a buffer manager module is required. If an external memory block is used then a limit to the number of queues which can be supported must be set. For each queue two blocks of information can be envisaged - queue state and datagram records. Datagram records are stored in a queue (circular buffer) of their own. The following parameters could be stored by the queue state and datagram record data structures in the buffer manager:

Queue state -	*Head and tail pointers* - required for basic queue management
	Datagram start address - required so that the tail pointer may be restored in the event of tail drop.
	Top and base address - Configurable parameters which define the limits of the queue (memory mapped buffers only)
	Restart threshold - Configurable threshold which introduces hysteresis into the buffer. Enqueueing may only restart when occupancy drops back below threshold.
	Queue occupancy - a running record of the number of datagrams in the queue
	Interrupt threshold - Configurable parameter which dictates the level of occupancy which is required to trigger an interrupt - typically 1.
	Target processor ID - Configurable parameter which selects a target for the interrupt.
D'grm records -	*Datagram size* - Modified as chunks are enqueued
	Datagram control information - 'Undefined' bit field of the Block specific bits extracted from first chunk header
	Datagram valid - A flag which declares that the record is complete (datagram fully enqueued)

This information could be used as follows. When a chunk of a new datagram arrives a new datagram record is created with the *Datagram valid* bit cleared. The 'Undefined bitfield' of the first chunk header are written to the record. As successive chunks arrive the *Datagram size* field is updated and the *Tail pointer* is incremented. When the last chunk of a datagram is enqueued the *Datagram valid* bit is set and the *Tail pointer* is copied into the *Datagram start address* field. If a queue overflows then further chunks are discarded and the *Datagram start address* is copied to the *Tail pointer*. Enqueueing may only restart when the occupancy drops back below the *Restart threshold*.

The buffer manager connection to the control interface has two purposes. When *Queue occupancies* exceed *Interrupt thresholds*, interrupts can be generated to the processors identified by the *Target processor ID field*. Interrupts are normally used in conjunction with the 'Conventional Read' mode of operation. As described in 2.1.2, an interrupted processor must read the *Head pointer* and *Datagram size* in order to ascertain how to recover the datagram from memory. At the same time the processor can read the *Datagram control information*. The datagram can then effectively be read from memory as a single chunk. When the read is complete, the processor must write a new *Tail pointer* value to the buffer state information and invalidate the record in the Datagram record queue.

PRODUCER: The producer services batch read requests from processors. These requests can be pipelined in order to maximise the rate at which the buffers may be emptied. The basic request instructs the producer to send *N* chunks of data, each chunk conforming to a specified maximum length. Chunk headers are generated by the producer using its own state information (about chunks it is sending and their relationship to a datagram) and chunk information recovered from the buffer manager[1]. If there is insufficient data in the buffer to satisfy the request then the request may be terminated prematurely. This event is recorded in the 'batch end' flag of the chunk headers. The conditions that accompany requests could typically invoke the following behaviours:

- Send all chunks unconditionally. This would effectively cause datagrams to be wrapped between separate requests. Any chunks of wrapped datagrams would have their chunk header wrap bits set.
- Do not split datagrams between requests. (ie. do not start sending chunks of a datagram if the whole datagram cannot be transferred within the specified *N* chunks.) The batch end bit is used when requests are terminated prematurely.
- Terminate batch if data queue becomes empty. Maybe there should also be a timeout on this.

A minimal set of conditions should be agreed upon which in combination provide the maximum versatility. This is a matter for further thought and discussion.

Although data buffers may not send unsolicited data to processors there is no reason why data buffers should not be allowed to send to other data buffers. The problem with this arrangement is that it apparently requires the producer to automatically send without request. Various schemes could be applied by the system designer. It could be arranged that a supervisory processor receives interrupts from a data buffer block and then issues the blocks producer with requests. Alternatively, if a data buffer block has a single buffer and a single output stream then a single request could be loaded into the producer on system initialisation which specifies an infinite chunk size and infinite number of chunks.

CONTROL INTERFACE: The control interface bundles together all control function. It provides:

- Access to the data buffer blocks hardware semaphore. This may be used by processors to arbitrate for the block. .
- An output for interrupts generated by the Buffer manager
- A means by which processors can access counters and registers. This may include access to tail drop reporting counters and buffer manager state information.

3.1.4 Interface blocks

Interfaces are simply 'straight through' entities which may exist between processors and data buffers. They often perform the basic function of converting datastreams between on-chip bus and inter-chip physical signalling protocols.

Another important role for interface blocks is to sit on the boundary between the application and system domains and perform chunk header addition and removal. As previously described, within the system domain all data streams flow in chunked form, each chunk having an attached chunk header. On entry to the system a complete datagram must be identified and an appropriate chunk header prepended. At a minimum the datagram start and end flags must be configured. The chunk length may optionally be defined as the datagram length if system processors require this information. Conversely, on egress from the system domain the chunk header must be removed.

[1] Storing chunk information in the buffer manager in this way instead of in the buffer memory is neat and tidy. Storing control information with data causes problems for both conventional read modes and for the producer if it were set up for proxy reads.

Interfaces are likely to contain application specific logic, therefore custom on-the-fly operations could also be carried out on data. Typically, such functions would add further information to the chunk header. For instance basic datagram authentication may be applied which could result in the chunk header exception or error bits being set.

3.2 NP1 Subsystem Overview

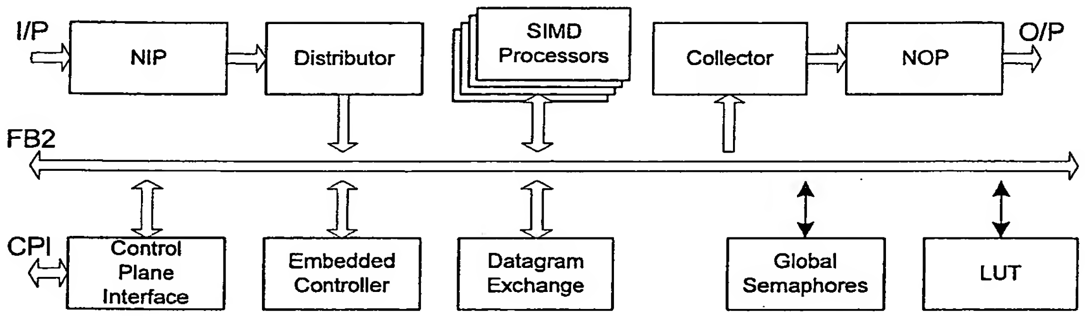


Figure 3.5: Principal components of the NP1 SoC

With the architecture organised in this way the main data paths through the chip are supported as follows:

- The distributor and collector are data buffer instances which handle the single fast path stream of data from the NIP and to the NOP (respectively). The distributor buffers incoming data from the NIP and supervises the transfer of data to the SIMD processors as requested by their thread sequence controllers. The term 'distributor' is derived from the blocks capability to forward incoming data to multiple processors. Conversely, the collector buffers data sent by multiple SIMD processors and multiplexes it into a single stream to the NOP.
- The NIP and NOP are application specific interfaces for use in network processor SoCs. They perform chunking and dechunking at ingress to and egress from the system domain, and provide a conversion between the physical signalling at the pins and a standard bus interface protocol (VCI). The NIP/NOP may connect to the Distributor/Collector via either a block to block interface or via a VCI compliant on-chip bus segment.
- The Datagram Exchange (DEX) block is a general purpose data buffer which acts as a shared hub via which data streams may flow from any source on the chip to any destination, provided that both source and destination connect to the on-chip bus. This provides enormous flexibility in the way that different on-chip data streams may be supported and provisioned under software control.
- The control plane interface provides a path for data between the off-chip control processor and the Datagram Exchange. It converts between the off-chip bus protocol (eg. PCI-X or RapidIO) and the on-chip bus. DEX provides the coupling between the fast and slow data paths.

3.3 NIP/NOP blocks

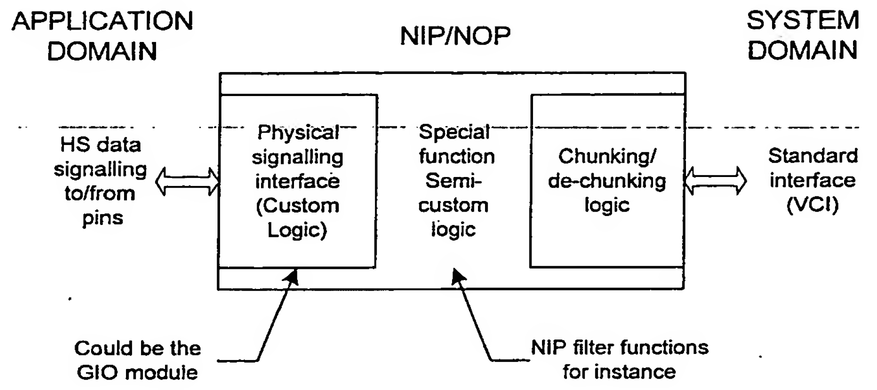

APPLICATION DOMAIN
NIP/NOP
SYSTEM DOMAIN
HS data signalling to/from pins
Physical signalling interface (Custom Logic)
Special function Semi-custom logic
Chunking/ de-chunking logic
Standard interface (VCI)
Could be the GIO module
NIP filter functions for instance

Figure 3.6: Basic organisation of the application specific NIP and NOP blocks

The primary function of the NIP and NOP blocks are to convert between the physical signalling protocols (eg. High speed serial) used in networking applications and the standard, synchronous protocol (VCI) used at interfaces of the PixelFusion sub-system. The NIP and NOP are therefore both types of the 'Interface' model. As shown in the system overview, the NIP and NOP both connect to the distributor and collector data buffer blocks. Because the distributor is a type of the data buffer model, it expects a prepended chunk to exist at the start of every arriving burst of data from the bus. There is a complication in that NIP and NOP can connect to either each other (pipelined NP1s) or third party system off-chip entities. The following are examples of potential approaches:

1. Prescribe that the NIP always adds chunk headers to every block of data received externally. Either using explicit signalling from an upstream framer chip, or implicitly from the external link data protocol and framing, the NIP must be able to identify the start and end of each packet and populate the chunk headers with appropriate information. This requires that the NOP always strips chunk headers from packets as they are output. The disadvantage of this approach is that should the system domain contain multiple PF NPx chips, no inter-processor transfer of control information is possible using the chunk header between processors in different chips. The advantage is NIP/NOP simplicity.

2. Prescribe that chunking and de-chunking are configurable options in the NIP and NOP. Chunk headers could then be retained in NOP to NIP connections. Although this is more complex it is preferable from a system design standpoint.

3. A third way is to design portfolio of NIP and NOP interface blocks for networking. If the global, system level architecture can clearly define the number of SoCs required then the SoCs can be constructed from the IP portfolio according to their location in the system. In other words NIP and NOP blocks are selected to create head of pipe, tail of pipe, intermediate or stand-alone NP1 variants as required.

It is anticipated that a whole family of different interface IP blocks may be designed by us and/or our customers to suit different second level interconnect (board level) protocols and support different filter functions. The functional scope and complexity of the interfaces is therefore deliberately kept to a minimum to simplify block reuse and customisation. All buffering function is contained in the more generic distributor and collector IP blocks. To simplify this redesign effort, the only functions which are permitted in the interface are those which are specific to the application domain. To put any such function in the distributor or collector would pollute them making them less generic and reusable without change. There is no obvious additional functionality required in the NOP at this time. The NIP however is required to perform coarse grained payload identification on the fly based on layer two information. Packets arriving on unsupported protocols must be identified and marked for forwarding to the control plane downstream. Additionally, there is value in a size filter which can identify packets which exceed a configured maximum size. These could also be diverted to the control plane (via the central memory_resource) or to an error_queue. The exception and error bits in the chunk header can be used for this purpose. The layer two information is inserted into the undefined bit subfield of the block specific bits in the chunk header to assist downstream classification. This function may need to be bypassed/disabled when a NIP is connected in a pipeline to an upstream NOP.

3.4 Distributor block

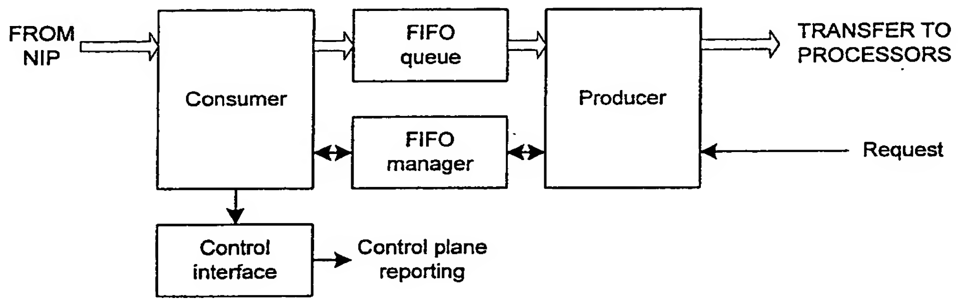


Figure 3.7: Schematic overview of the distributor block

The Distributor is a type of data buffer block. The modules have the same functions and interactions as described in section 2.1.3. Tailoring specific to the distributor is as follows:

- There is only one stream from the NIP therefore a single buffer can be implemented as a dedicated FIFO embedded within the distributor. The FIFO manager is a single queue implementation of the buffer manager.
- It is assumed that backpressure to the NIP is not permitted. Tail dropping from the FIFO is therefore supported and byte/packet drop count information is made accessible to control plane software.
- Because the distributor is a block of IP designed specifically to buffer a single high speed data stream two of the control functions shown in the general model do not need to be implemented. As traffic arrives from a single source no hardware semaphore is required for the block. Furthermore, it is intended that processors access the distributor using only batch reads. The interrupt and processor interface to buffer manager information is thus not required.
- It is generally prescribed in this document that a data buffer should never send data to a destination unsolicited. However, this specifically refers to sending to processors. It would be permissible to send to another data buffer entity. The main function of the distributors producer module is of course to forward data from the FIFO packet store in compliance with batch read requests. During the course of this process, packets may arrive at the head of the FIFO which have been marked for special treatment using the Exception and Error bits of the

chunk header. These packets should be forwarded on the bus to an alternative destination identified by a configured address. This function is deemed to be a useful generic attribute for the distributor to have and would occur transparently during the servicing of batch read requests. The application specific decision making and marking is of course a function of the upstream application interface (NIP). The distributor can thus be configured to intercept on the fly both jumbo packets and packets with unidentified protocols and send them to exception queues in the Datagram Exchange. From here they can be processed by a dedicated exception processor or by the control plane processor (for instance).

- Regarding IP deployment and scalability, the FIFO queue size should be configurable within defined limits. The minimum size could be derived from the configured maximum application datagram size (64k IP packet in this application).

3.5 Collector block

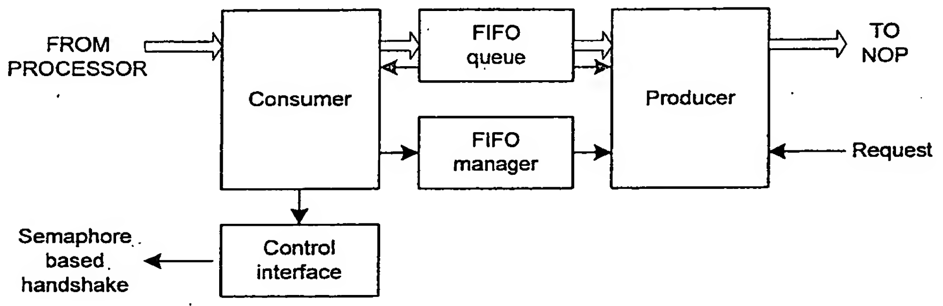


Figure 3.8: Schematic overview of the collector block

The Collector is a type of data buffer block. The modules have the same functions and interactions as described in section 2.1.3. Tailoring specific to the collector is as follows:

- The collector, like the distributor, is a block of IP designed specifically to buffer a single high speed data stream. A single stream is therefore buffered by a dedicated hardware FIFO. As complete packets are auto-forwarded in batch mode to the NOP the interrupt facility and processor interface to buffer manager information are not required.
- The basic producer module is configured for auto-forwarding by initialising a request into the request queue which specifies that an infinite number of chunks of infinite size should be sent unconditionally. Batch termination should be turned off.[2]
- System level arbitration for the shared collector employs a global semaphore. This semaphore is implemented within the collector block. This is because there is no guarantee that the pool of general purpose semaphores (Semaphore block) will exist in a system. In order to reduce the delay when one processor hands ownership of the collector to the next, the semaphore is accompanied by a request queue. Multiple processors may wait on the collector simultaneously and be serviced in turn.

[2] Interestingly, since the distributor also uses the same producer module the distributor could be set up via the control plane to auto-forward all traffic to the collector. This may be a handy test mode which could be used to bypass the SPs.

- The consumer uses the blocking action of the bus to signal congestion upstream to the SIMD processor. There is no tail dropping from the FIFO and hence no reporting to the control plane is required.

3.6 Datagram Exchange block

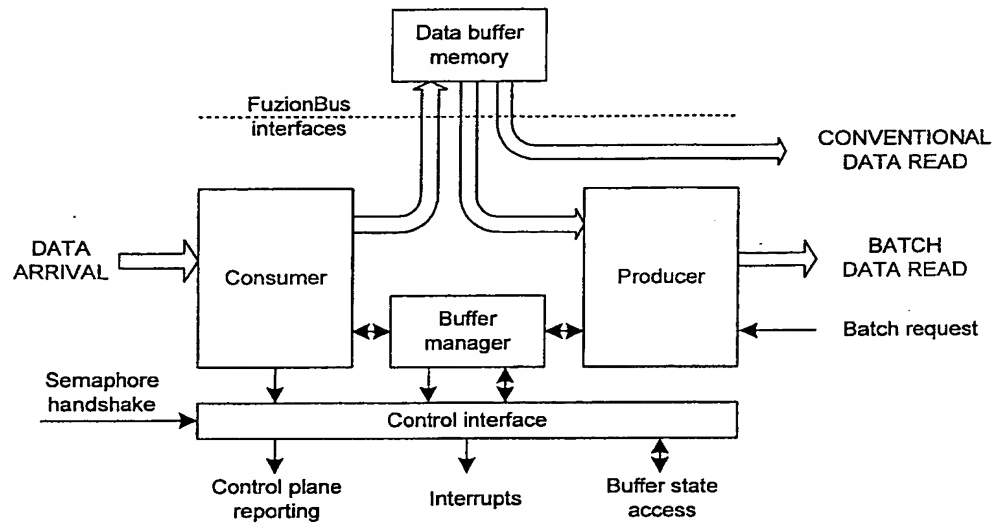


Figure 3.9: Schematic overview of the Datagram Exchange block

The Datagram Exchange is a fully endowed implementation of the data buffer block model. It has already been established that wherever one processor produces data which is consumed by another a data buffer is required in order to decouple the different data rates and data chunk sizes. Furthermore, multiple buffers are required in order to resolve contention whereby multiple processors feed data to a single processor, or where data output from a single processor needs to be segregated (purgatory). Many such data paths may exist in a system. It is impractical to provision every path with buffering resource in hardware. The purpose of the Datagram Exchange is to provide a hub via which any path between processors may be routed. This simplifies the design of other blocks in the system and enables software to efficiently allocate resource only to the paths which are required.

The Datagram Exchange is especially applicable to the data paths which exist between the fast and slow path, and to the global recirculation paths which could be applied to packet purgatory. There are a multitude of approaches that the system designer or programmer may take in dividing up and logically interconnecting the various processing resources. The Datagram Exchange allows these decisions to be made at application design time, and not forced at hardware design time. The architect designing a general purpose system simply provides the DEX via which any arrangement of paths could potentially be configured, a suitable quantity of data buffer memory, and a decent portion of bus bandwidth in support. The main features of the DEX are as follows:

- A bounded number of queues may be configured into the memory space. Queue state is maintained by the buffer manager as described in section 2.1.3. As a general rule[3], a queue is configured for each unique logical data path from a processor block (such as a SIMD processor) to the DEX. This avoids contention between producers trying to send to the same resource without having to use global semaphores for arbitration. The system remains simple. A general pool of memory can thus be allocated according to which system capabilities need to be supported. This is an efficient, flexible, IP friendly approach.
- The DEX consumer is similar to the consumer in the distributor. It must enqueue data, count bytes, record packet size, manage buffer overflow and report dropped packets. The main difference is that the data buffer is not a dedicated FIFO but a block of addressable memory which can support multiple queues. The interaction with the Buffer manager block is therefore more complex as state for multiple queues must be maintained and packet chunks destined for different queues may arrive in an interleaved fashion.
- The data buffer memory is an independent block of memory addressable via the bus which could be on or off-chip. This supports conventional read accesses from processors.
- The producer is essentially the same module as used in the distributor block. The only possible variation is that a proxy DMA function could be added. For instance, with data memory probably (though not necessarily) located independently on the FuzionBus, when the producer services requests it could provide the module ID of the requesting processor instead of its own. This means that the read data is sent directly to the request originator and not back to the Datagram Exchange block. The VCI interface can support this function.
- The control interface module is fully featured with: a hardware semaphore which may optionally be used by blocks contending for the central memory resource, dropped packet reporting to the control plane (RMON), interrupt generation and buffer manager state access for conventional read accesses.

3.7 Control Plane Interface (CPI)

The control plane interface sits on the boundary of the PixelFusion IP subsystem. Its job is to translate control and data traffic between our system and that of the control processor and its associated system. Although the control processor system could take a number of forms, we can make the assumption that it consists of one or more CPUs which are connected to our system via something like PCI-X, RapidIO, LDT etc. Our CPI is then an interface block type with two sides, both of which must conform to its own standards.

Since the control processor cannot (by definition) deal with the data rates on the fast path data plane, some transfer negotiation or flow control must be implemented - again conforming to the system standards on both sides. We cannot treat the control processor as a black hole which just absorps everything we throw at it.

[3] The exception may be packet purgatory whereby SPs may wish to output exception packets to a single purgatory buffer (or even to one of a number of 'purgatory bins'). The mutual exclusion provided by the collector semaphore should also indirectly provide mutual exclusion between SPs writing to the purgatory buffer. If this is considered to provide insufficient protection then the DEX global semaphore could be used in addition to the collector semaphore to ensure more robust access to purgatory buffers.

On our side the CPI conforms to the model we have of an interface block. It simply passes transactions from one side to the other, translating the formats in the process. On the control processor side it conforms to whatever bus standard or interface exists between it and the CPU.

Flow control and data rate conversion is achieved by having the slower agent (the control processor) initiate data transfers to or from a shared memory resource in our subsystem (the DEX). When exception packets need to be passed to the control plane, the packet is deposited in the Datagram Exchange block and an interrupt sent through the CPI to the control processor, which can then read the packet via the CPI. A similar process is followed for the reverse process. All data communications goes via the DEX to decouple transfer time and transfer rate between the two sides.

3.8 SIMD Processors (SP)

The SIMD processor block is described in conventional terms as indicated in Section 2.1. Control interaction with other processors and hardware blocks is performed via semaphores. The processor issues read and write instructions to Distributor, Collector and LUT etc. A range of addressing modes are typically required for these accesses where PE's are concerned. The processor may be a target for interrupt requests, but the processor's memory (neither controller or PE's) is never a target for read or write operations issued from another processor or block. (In a future roadmap you could imagine SIMD processors in a NUMA configuration, but it is unnecessary and undesirable for this kind of application).

When the processor issues read and write requests, the timing of the data transfer is determined by the transaction target. For read operations, the VCI bus supports split transactions so the target responds with data when it is ready. For writes, the bus system's built in flow-control will throttle transfer as the target accepts it.
By programming a pipelined semaphore scheme between processors, much of the control latency can be removed. This is described in detail in sections below.
The SIMD processor will perform both conventional and batch style data transfer operations. Reading from the packet input stream and writing to the output stream are obviously batch style since packets are datagrams and we are transferring a number of packets in the same operation. Packet transfer between processor and DEX is also batch based. Access to the peripheral blocks such as LUT is a conventional memory style access. All read and write operations performed by the SIMD controller (e.g. instruction fetch) are also conventional style.
In the currently proposed SIMD processor design there are I/O ports on the PE array termed DirectIO (DIO) and BlockIO (BIO). These map neatly onto the two transaction models described here – batch mode deals with datagram chunks and is supported by the DIO mechanism, and conventional mode is supported by the BIO mechanism. One difference in this proposal is the removal of the DIO 'slave' mode in which other blocks in the system can directly invade PE memory. In this architecture (and all others that we can think of) processors are the masters of their own memory space.

3.8.1 PE I/O requirements

Getting data in and out of the SIMD processor is a central function of the architecture. Achieving software flexibility while maintaining price/performance (lots of bandwidth, minimum storage) needs a careful design. Let's consider what it must do:

1. Support an amount of memory (the chunk size) in each PE which is used for data input/output. Each PE may receive a whole datagram (any data of variable length - lets call it a packet) if it is smaller than the chunk size, or part of a packet, if bigger. It never receives more than one packet. The DIO data buffer defines the chunk size of the batch style external data access.
2. Support read instructions, where those PE's that wish to receive new data can do so.
3. Support write instructions, where those PE's that wish to transmit new data can do so.
4. Maintain the integrity of packets which are spread over multiple PE's.
5. Support batch style & conventional style access to external blocks.

Some second-order requirements follow on from these main functions, because of the way we are using SIMD to handle variable length data:

1. Fragmentation of available PE's is bad. Maintaining the integrity of a packet forbids spreading it over non-contiguous PE's. The array will much more efficiently used if all PE's requesting new data are consolidated together for allocation to incoming packets. This introduces the idea of 'recirculation', whereby packets retained by any subset of PE's for further processing are all shuffled along to one end of the array. Since all PE's are the same, moving the packets from one set of PE's to another makes no difference as long as all the intermediate state goes with them.
2. When a packet is smaller than the PE chunk size, some of the memory of that PE will not be filled, and is essentially 'lost' to the read instruction. This means that the target which the data is read from needs to know how many PE's want data and what their chunk size is, rather than be requested simply for the total amount of available memory that would include this 'lost' space. Otherwise, it may return more packets than can be accommodated by the array. This requirement is covered by the definition of the batch style memory access in which both the number and size of requested data chunks is specified.
3. When many PE's request data in a read instruction, the target may not have enough data to satisfy them all. It should be able to terminate the response transfer prematurely, and have all PE's that missed out on data be notified, by receiving some kind of null response. This requirement is covered by the definition of the prepended chunk header word, which contains fields for indicating both the length of each chunk and the terminal chunk of a batch transfer.
4. Only some PE's with data will require to participate in conventional style I/O instructions. For instance, only those PE's with packet headers need to access the LUT block.
5. When executing a write instruction, PE's may not know whether to output their data, because this decision has been made on another PE. An example would be a packet which can simply be deleted from the stream, on the basis of its header contents. The other PE's with the payload data are not aware of the decision (unless by swazzling, but wastes many cycles on arbitrarily long packets).

3.8.2 Mark bits & chunk header bits

There are numerous ways to meet the above requirements. An example for the batch style I/O is outlined here (but not all the details).
Each PE has a 'mark bit' which indicates to the I/O hardware (DIO transfer engine) the PE's participation in a read or write instruction. This is familiar stuff from F150 etc. We also use two of the eight bits in the chunk header word that are reserved for private use within blocks (see Section 3.1.1). We term these two bits:

Output
Recirculate

The use of these fields is explained by an example where the SIMD processor uses batch style reads from and writes to buffer type blocks (such as Distributor & Collector). The system address of the target blocks may best be supplied by the controller rather than be replicated in every PE. The case where PE's supply individual addresses would be a different addressing mode and not described here. First some terminology:

N = # of PE's in array
n = # of PE's participating in an I/O operation.
S = nominal chunk size handled by a PE.
s = actual amount of data a PE has, where s ≤ S.

Now consider the case where the controller knows that all PE's wish to receive a data chunk from a batch read instruction (as would be the case the first time such a read instruction is issued). In this case the mark bits are not required, and the controller issues a batch read request for "N*S" bytes of data (the individual values N and S are contained in the read request). The target buffer block returns up to N chunks of data, each of maximum size S. A control word is supplied at the head of every chunk. The first and last chunks of any packet spread across p PE's have the DatagramStart and DatagramEnd bits set respectively, as in the following table:

PE # in packet	0	1	...	p-2	p-1
Mark Bit	X	X	...	X	X
DatagramStart	1	0	...	0	0
DatagramEnd	0	0	...	0	1

Now the PE's process the data, perhaps changing the lengths of their chunks within predefined limits. Most PE's with headers will simply wish to forward their packet to its destination. Some PE's may decide that their packet is to be discarded. Others may contain a tunnelled packet which should not be output from the processor until another round of processing has been performed on it. Still others may decide that their packet needs multicasting, so that it should be written out, but also retained for the next round of processing so that it can be written out again to a different destination.
When the time comes to write out the packets, the format of the mark bits and prepend word bits across p PE's containing the chunks for a single packet is as follows:

PE # in packet	0	1	...	p-2	p-1
Mark Bit	1	0	...	0	0
DatagramStart	1	0	...	0	0
DatagramEnd	0	0	...	0	1
Output	[1:0]	X	...	X	X
Recirculate	[1:0]	X	...	X	X

Here, the mark bit denotes not that the PE has valid data, but that it has the *first* chunk of a valid datagram or packet. This is duplicated in the DatagramStart bit of the chunk control header. The last chunk has its DatagramEnd asserted. These header bits will be exactly the same as when they arrived, unless the number of chunks in the packet has been decreased during processing (it cant be increased!). The two additional bits have been set by the PE software and indicate to the transfer engine what is to be done with this packet. They are only significant in the first chunk. For the example packet types mentioned above the bits are set as follows (the discarded packets do not have mark bits set, and so their prepend bits are irrelevant):

Prepend Bit	Output	Recirculate
Normal forward	1	0
Detunnel	0	1
Multicast	1	1

The transfer engine scans the PE array until it finds one with the mark bit set. This indicates the start of a new packet to transfer. It reads the data chunk, and decides what to do with it based on the two new header bits. It can send it out as part of the write operation, recirculate it back to the array, or both. Once a chunk is encountered that has DatagramEnd set, the transfer engine knows that the packet is complete and goes back to scanning for the next mark bit that is set.
The transfer engine has to maintain two 'pointers' into the PE array – one for scanning for mark bits, and the other for the tail pointer for the chunks being recirculated back into the array. It also maintains two hardware counters, TotalOutput and TotalRecirculate, and at the end of the write operation the controller is informed (or may access) how many chunks went either way. There could also be a third counter for total number of packets if that was useful.
The recirculation is performed as a pipeline process on parallel data paths to and from the PE's. It should require no more data storage than would already exist for input and output (approx a chunk's worth).
On the next batch read instruction which will deposit data in the PE memory buffer to which packets have just been recirculated, only n PE's have space for new data (N-n already have some old). The controller knows how many because it has the TotalRecirculate count from the previous write instruction, and is able to issue a read for "n*S" bytes of new data.

3.8.3 Rechunking of Recirculated Packets

Since a processing round may increase or decrease the lengths of chunks in PE memories, it might be desirable for packets to be rechunked into the nominal chunk size as they are recirculated in the PE array. That would make them look as if they had come fresh from the input stream, and may allow a more efficient allocation of PE memory. This function could be included in the SIMD processor if necessary.

3.8.4 Address modes

In the above description we have only considered an address mode for reads and writes, suitable for batch style access, where all PEs share a common external address, which may be supplied to the transfer engine by the SIMD controller. This We might denote these mono address, poly data modes like this:
read addr, n*S
write addr, n*s, n*data
Where PE's read or write to individual addresses, as is the case for conventional style memory access, you have poly address, poly data modes like this:
read n*(addr, S)
write n*(addr, s, data)
There is also a requirement for the transfer engine to be able to issue single conventional style read and write transactions to an external address, for the purpose of implementing high speed semaphore based access control to a shared resource. More details in the section on operation below. These are essentially read and write instructions executed by the transfer engine, for which the PE's do not participate at all! We may donate them as:
read addr, 1
write addr, 1, data
Other addressing modes may likely be desirable, as variations of these.

3.9 Global Semaphore Block

Coordination of the different processors in the system is achieved through software by means of shared hardware semaphores. Where these semaphores are associated with particular hardware functions, they can be contained within the corresponding hardware block. For general purpose sempahores which software uses to synchronise system operation at a logical level, a block of semaphores is provided as a stand-alone unit connected to the system bus. The details of this peripheral type unit are as described elsewhere.
No new bus transaction types are required to use the semaphore block. The semaphore operations are mapped to bus transactions in the following table.

Semaphore Op	Bus Transaction	Comment
Signal	Write	May carry data to attach to signaled semaphore
Wait Request	Read Request	Requesting unit waits for response from split transaction bus.
Wait Acknowledge	Read Response	Semaphore unit responds with data that was attached.

Setting a semaphore and passing some data via that semaphore is a write operation.
Waiting on a sempahore and picking up its associated data is a read operations, with the waiting function implemented via the split transaction nature of reads on the bus.

3.10 Plugin Blocks

A set of specific hardware plugins can be added to the basic architecture to assist with particular applications. These are optional 'peripheral' type blocks and do not disrupt the general design of the system. They could be accessed by batch style or conventional style transactions. Some examples for network processing listed below.

3.10.1 LUT

The lookup table block is as described elsewhere. Essentially it is a special purpose memory controller block, which provides front-end logic to implement a specific memory related function. The memory is typically divided into multiple banks. These banks may be embedded in the LUT block itself, discrete memory blocks located on the common bus, or external to the chip. The mapping here is all a matter of bandwidths and latencies.

3.10.2 Counter Block

For handling many thousands of counters in a network processor there has been a proposal for a special purpose accelerator block, consisting of memory fronted by dedicated logic. Very similar to the LUT block in effect.

3.10.3 String Search Block

Another potential memory + special logic function block. Can you spot a pattern emerging?

4 System Operation

4.1 System configuration and behaviour

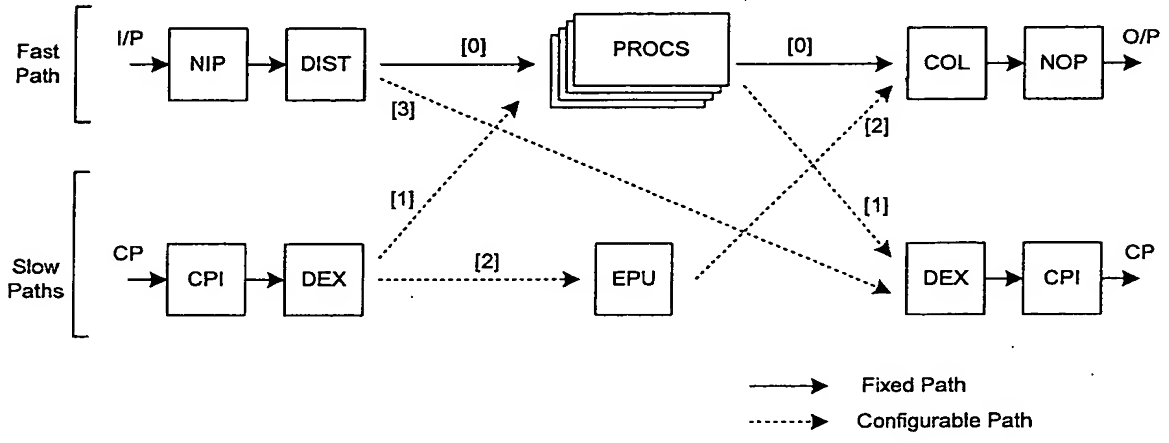


Figure 4.1: Overview of main datapaths between system components. Note that MEM and CPI are in fact single blocks. The are split in the diagram in an effort to improve clarity.

The purpose of this section is to clarify how the component blocks described in the previous section could interact in different configuration scenarios. (This is not an exhaustive list). The solid arrows represent data paths which have fixed hardware support and are assumed to exist in the basic system architecture. For instance, the distributor provides a dedicated buffer in hardware for fast path forwarding of data from the NIP to the SPs. The broken arrows represent all the data paths which are supported by the architecture but need not be configured. ie. a physical connection over FuzionBus is available but memory need not be allocated for path buffering in the Datagram Exchange if the path is not required. Numbered data paths are described below:

[0] This is the basic fast path. SPs use global semaphores to load from the distributor and unload from the collector.

[1] Basic slow path options. These can be divided into two basic types:

(a) For interfacing to the control plane, SPs could load from a control plane input buffer or unload to a control plane output buffer in the Datagram Exchange. At this time little provision is made to enable a split load from both the distributor and the control plane input buffer. It is possible to do; however, it is assumed that a SP will periodically load from DEX (based on an interrupt) instead of the distributor when the control plane input buffer becomes relatively full.

(b) For global recirculation the SPs output to recirculation buffer(s) configured into the Datagram Exchange. The recirculation buffers are subsequently emptied by the SPs on interrupt. Global recirculation in this way is specifically intended for packet purgatory.

[2] Slow path injection bypassing SPs. Control plane traffic loaded into a control plane input buffer in the Datagram Exchange is served by the EPU instead of a SP. The EPU would use global semaphores to gain ownership of the collector and then request that DEX send N chunks directly to the collector (wrapping off).

[3] Slow path interception bypassing SPs. Traffic arriving at the NIP is classified according to layer 2 information (layer 3 protocol). If the content is not-supported (not IP) then the packet may be marked. A generic attribute of the distributor then comes into play which automatically forwards marked packets to a specified address (the control plane output queue in the Datagram Exchange). This does not affect the normal servicing of requests by the producer in the distributor. The control plane output queue is emptied by requests arriving from the control plane processor.

4.2 System control with global semaphores

Controllers (such as SPs) share common resources such as the distributor and collector blocks. In order to resolve contention for ownership of such blocks global semaphores are used. The global semaphores effectively provide a system level arbitration mechanism under software control which works with (but is independent of) the on-chip bus in which all arbitration is distributed. The advantage of software control is that arbitration is very flexible. The perceived cost of this flexibility is that there is a delay when SPs are granted ownership of, and later release shared resources. If this delay is too large then the long term average rate of data forwarding drops below the data arrival rate thus causing data to be lost.

This section examines methods by which control latencies can be reduced, and identifies further issues introduced by the measures.

4.2.1 Analysis of problem

What is the available time (on average) between loading or unloading each SP during which control messaging may occur? The following points which refer to loading may be equally applied to unloading.

- Assuming a stream of minimum sized 40 byte packets arriving at 5 GB per sec, approximately 1.5M SIMD buffers must be loaded per second. This assumes 64 PEs per processor, each loaded with a 48 byte chunk. It also assumes virtually no overheads from layer 1 and 2 processing.
- On-chip, the bus and raw DIO bandwidth is 6.4 GBytes per sec. (This assumes ideal, lossless DIO streaming). Since buffer memories in the distributor (and collector) blocks decouple the off and on-chip data streams, a delay may occur between each SIMD load phase.
- If SIMD loads are spaced evenly over time, the delay is calculated to be 160 ns, or 64 cycles of the 400 MHz clock.

Applying the same approach to large packets arriving and completely filling PEs the delay between SIMD loads is approximately 286 cycles.

Let us now consider a typical sequence of events which may occur when a SP uses global semaphores to gain control of a shared resource on the bus. The resource is made available to the SPs on a round robin basis.

1) SP waits on the resource by sending a FuzionBus read request to its semaphore in the global semaphore block.
2) The semaphore is signalled by the previous SP in the RR sequence (FuzionBus write request), causing a read response to be returned.
3) The SP may now initiate the data transfer between its array and the external resource. This may involve FuzionBus control messaging, set up of the DIO block, and most importantly transfer of data between the DIO memory and the PE memory.
4) When the operation is complete, the SP can signal the next SPs semaphore thus effectively handing over ownership of the resource.

The delay incurred by inter-SIMD resource handover is then represented by the time taken for events (4), (2), and (3) to occur (in that order). When bus transactions, block logic and most importantly PE memory to DIO memory transfers (ranging from 12 to 64 cycles - ignoring compute behaviour) are accounted for, the delay could be of the order of 40 to 90 clock cycles.

The earlier estimates on available budget were 64 to 286 cycles. However, in a hypothetical (though fairly realistic) scenario whereby traffic may comprise a significant proportion of small packets interspersed with moderately sized packets we get the worst case - the small packets act to reduce the gap between unloads while the larger packets increase the control latency. The budget is therefore likely to be nearer the lower end of the given range. This is clearly too close for comfort.

A further exacerbation of the problem occurs when we consider how a SP reserved for exception packet processing contends for the collector. This exception SP may not require the services of the collector on every round; however, it must still be included in the round robin sequence so that it may be polled. If the exception SP does not require its turn then it simply reads to clear its semaphore and sets the next. This adds further cycles to the delay between the outputs from the SPs before and after it in the sequence.

A solution is required which retains the flexibility provided by the global semaphore scheme for controlling SP resource hand-over whilst reducing the time taken between SP data transfers. ie. some of the control signalling must occur concurrently with data transfer.

4.2.2 Solution for load from distributor

The distributor sends data to destinations on request. There is no need to use global semaphores to manage requests arriving from independent controllers. A simple request queue suffices. The distributor processes each request in turn with very little delay in between as long as the request queue is not empty. Mutual exclusion is built in.

The problem lies with the use of global semaphores to lock SPs into a round robin sequence. Using only one semaphore per SP results in a delay between one SPs load completing and the next sending out its request.

To built request pipelining into the round robin sequence 2 semaphores per SP could be used as described below. The example is for a system with 4 SPs sequenced round robin in the order A,B, C, D, and a request pipeline 2 deep.

- Semaphores As, Bs, Cs and Ds are used to control the sequencing between SPs.
- Semaphores Ap, Bp, Cp and Dp are used to control the pipeline depth

Lets say SP C wants to initiate a NIP-to-array transfer:

1. SP C TSC waits on Cs (ie. until SP B in the sequence has sent a request and signals Cs)
2. SP C TSC waits on Bp (ie. until the load to SP A has been completed and there is therefore room in the distributor request queue)
3. SP C TSC issues request to distributor to perform a data transfer
4. SP C TSC signals Ds
5. Distributor to SP C transfer occurs some time later
6. SP C TSC signals Dp

The pipeline depth can be controlled in software using the same 8 semaphores in the same way. The only change to the sequence shown above for a depth of three would be that at step 2, TSC would read the pipeline semaphore Ap instead of Bp. Obviously this is pretty limited by the number of SPs used and due to the DIO configuration issue described below it is unlikely to be desirable to pipeline more than two requests.

Discussion point: The problem with this approach is that a SP must set up its DIO for input before pipelining the request. If the request takes a long time to be serviced then the DIO is unable to be used for output from a different memory bank in the meantime. This may impact on performance in certain cases and is probably a candidate for modelling.

4.2.3 Solution for unload to collector

Two semaphores per SP can be used as described in the last section to provide pipelining and reduce the delay between two SPs unloading to the collector. In other words, as one SP is unloading, the next must be setting up its DIO and ULE in readiness to start. Global semaphores as described simply identify who is next. They do not control the actual access to the collector.

A feature must be added to the ULE control logic to enable it to communicate over FuzionBus with a global semaphore in the collector when it has been set up to unload by the TSC. ie. Once the SP has been given permission to unload next, it waits on the collector semaphore until whichever controllers using the collector before it have completed. Thus, at a higher level the TSCs retain system control and enable round robin sequencing to be controlled in software using the general purpose global semaphores, whereas at a low level the actual data transfer is controlled efficiently by the ULEs using the hardware based collector semaphore. The two are independent and latency is reduced to the time taken for a single message to propagate over the bus.

5 Development Roadmap

The architecture can be developed in several stages. At each stage functionality is added in hardware and/or software. These additions do not cause disruption to what was there before.
In discussions with the team, there seems to be a consensus that wrapping packets across more than one SIMD processor is a bad thing. In the very simple case it can work, but trivial cases are not interesting and in all the cases where you actually want to process the packets it leads to all sorts of problems. In the following roadmap, wrapping across processors is illustrated as a capability of the architecture - however, the authors personally prescribe to the view that wrapping should never be permitted. The roadmap ultimately leads to a situation in which wrapping is unnecessary.
This raises that perennial question of how to deal with jumbo packets - ones that are nearly as big, or bigger, than the aggregate size of data that the PE array can load up in one go. The answer is probably itself part of the development roadmap - initial versions of the architecture will simply not deal with packets to big to load, and future enhancements to the SIMD processor may be able to handle this.

Matrix summary

Stage	Fast path	Control Plane	Exceptions	Purgatory	Layer 7
1	Yes	No packet flow	No	No	Restricted
2	Yes	Yes	Handled by CP	No	Restricted
3	Yes	Yes	Yes	No	Restricted
4	Yes	Yes	Yes	Yes	Restricted
5	Yes	Yes	Yes	Yes	Yes

The features supported in the following sequence of stages is on the whole cumulative.

5.1 Stage 1: Fast path only

SUMMARY: In its simplest embodiment the system handles 'normal' packets only up to the maximum packet size. These require processing on the header only, and the header may grow or shrink. There are no exception packets that require 'special' processing or passing to or from the control plane. This basic configuration allows characterisation of the normal fast path.

ARCHITECTURE: The system includes the NIP, Distributor, SIMD Processor(s), Collector and NOP. The SPs batch read from the distributor with packet wrapping enabled. A control plane interface is also included but does not carry data traffic.

SUPPORTED FEATURES:

- Basic forwarding of all packets (including jumbo packets) belonging to supported protocols.

- Restricted layer 7 processing on all packets. (eg. string search only)

5.2 Stage 2: Control plane

SUMMARY: Adding a control plane enables exception packets to be handled

ARCHITECTURE: A Datagram Exchange block is added to the architecture. This provides the following paths across the system:

- Distributor to DEX - For filtering of illegitimate packets to the control plane.
- Processor to DEX - Processed packets sent to control plane
- DEX to collector - Control plane packet injection into fast path

Processors read chunks from the distributor in batch mode with wrapping disallowed.

SUPPORTED FEATURES:

- Unsupported and jumbo packets filtered to control plane prior to the SPs.
- Exception packets identified by SPs sent to control plane for processing.
- Control packet insertion into fast path downstream of SPs
- Processed exception packets reinserted into fast path downstream of SPs.
- Full layer 7 processing possible as no packet is split between any two processors (limited to small packets). The interpretation of 'full' is to enable routing decisions to be made based on payload content.

5.3 Stage 3: Internal recirculation

SUMMARY: The addition of internal recirculation capability to the SP greatly reduces the burden of exception packet processing on the control plane.

ARCHITECTURE: The function of the DIO system will need to be extended in order that chunks may be retained in the PE array during an unload operation. Refer to section 3.8.

SUPPORTED FEATURES:

- All normal sized exception packets which require reprocessing (such as multicast, de/tunnelling and ICMP packet generation) now handled internally within SPs. Large packets still filtered to the control plane.

5.4 Stage 4: Globa l recirculation

SUMMARY: Global recirculation enables all packets to be passed through the SPs regardless of size in order to preserve order in all normal packets. Large exception packets can be recirculated externally back to the SPs. Small exception packets can be recirculated either internally (more efficient) or externally (better order) as desired. Layer 7 processing would have to be restricted on large (wrapped) packets.

Global recirculation by definition also supports packet purgatory. This should be applied with care though. There are two separate approached which depend on how jumbo packets are being supported. If jumbo packets are being routed directly to the control plane by the distributor then small exception packets output from the core may be written directly to a number of purgatory queues in the datagram exchange. If jumbo packets are being routed through the SPs then all exception packets and packets intended for recirculation must be marked using the chunk header exception and error bits (perhaps a new global recirc bit could be defined). These packets are then intercepted by the collector and sent to a purgatory or recirc queue in the datagram exchange. Thus, in this mode only a single purgatory buffer could be implemented.

ARCHITECTURE: Global recirculation introduces the following new paths in the hardware:

- Collector to DEX - The collectors producer module is functionally the same as the distributor. Packets may therefore be marked in the SPs for interception and automatically forwarded to a configured queue in the Datagram Exchange. The marking of system flags in the chunk header in this case relates to recirculation. Obviously this requires that the Collector to NOP connection is via a FuzionBus segment and not direct block-to-block.
- DEX to SP - The SPs could select to read from DEX instead of from the distributor on the receipt of interrupts. The interrupts could signal SP semaphores which wake up exception threads in the thread manager. Software enhancements will be required to ensure that multiple SPs act on the same interrupt so that globally recirculated jumbo packets can be correctly read from the DEX buffer.

SUPPORTED FEATURES:

- Packet purgatory - single or multiple purgatory buffer support
- Jumbo exception packet support in the fast path

5.5 Stage 5: Full la yer 7 support

SUMMARY: The holy grail is for no packet to ever be processed separately by different processors. In other words, the ideal is to develop a SP architecture which enables any single packet to fit into a single SP. This eliminates any requirement for complex signalling of control information between processors (SPs) and enables complex layer 7 operations to be performed on all packets.

ARCHITECTURE: Will require a review of the organisation of the following scaling axes - PE numbers, PE memory size, DIO memory size. It will also require a detailed examination of how internal recirculation of large packets can be handled effectively and how provision can be made for jumbo packets without causing overprovision of resources when processing small packets.

SUPPORTED FEATURES:

- Full layer 7 support for all packets with no restrictions.